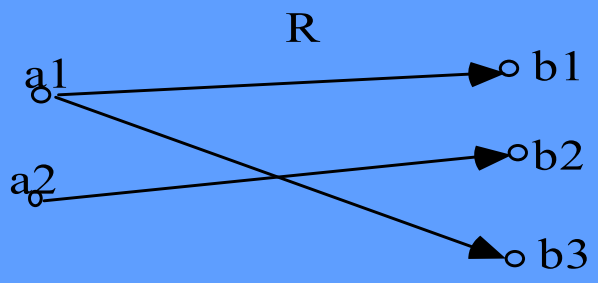
STRUTTURE DATI: OLTRE LE LISTE

- **Le liste risolvono un ampio ventaglio di problemi, ma sono strutture dati *sequenziali* e come tali *inadatte* a gestire *grandi quantità* di dati quando le operazioni più frequenti implicano un accesso *non sequenziale*.**
  - ad esempio, in una lista, la *ricerca di un elemento* richiede un tempo proporzionale alla lunghezza della lista: se essa contiene migliaia di elementi, tale operazione è inefficiente.
- **La gestione di *grandi quantità* di dati può avvantaggiarsi dall'adozione di strutture dati *non sequenziali***
  - ad esempio per rendere più veloci alcune operazioni frequenti e particolarmente rilevanti, come la *ricerca* di un elemento.

GRAFI

Un *grafo* è una struttura che rappresenta *relazioni binarie* su insiemi di elementi.

Una *relazione* fra due insiemi A,B è un sottoinsieme R del prodotto cartesiano A × B.
Qui A = { a1, a2 } e B={ b1, b2, b3 }.



Un *grafo orientato* è una struttura G = ⟨A,R⟩ dove

- **A è un insieme non vuoto di *nodi***
- **R è un insieme di *archi orientati*** tali che se $\langle a_i,a_j\rangle \in R$, c'è un *arco orientato* da a_i verso a_j.

GRAFI: NOMENCLATURA

- **Grado di ingresso del nodo a_i:** è il numero di archi che hanno a_i come nodo *finale*.
- **Grado di uscita del nodo a_i:** è il numero di archi che hanno a_i come nodo *iniziale*.
- Un grafo si dice **completo** se $\forall a_i,a_j \in A$ esiste un arco da a_i verso a_j.
  Se un grafo è completo, significa che la relazione R coincide col prodotto cartesiano AxA, ossia è una *relazione totale*.
- Si dice **cammino da α a β** una sequenza di nodi a_0, a_1...a_N tale che N0, $a_0=\alpha$, $a_N=\beta$, e $\langle a_i,a_j \rangle \in R$ per ogni $0 \leq i \leq N-1$.
  - cammino semplice: i nodi del cammino sono distinti
  - cammino ciclico (o *ciclo*): quando $\alpha = \beta$
  - cammino aciclico: se non vi sono cicli.

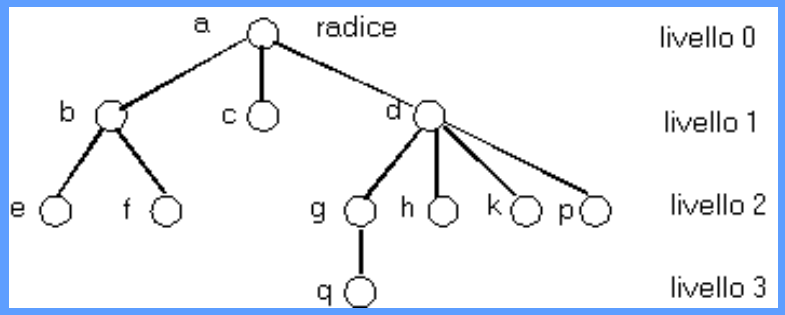
GRAFI e ALBERI

Un *albero* è un *grafo orientato aciclico* tale che

- **esiste un nodo (*radice*) con grado d'ingresso 0**
- **ogni altro nodo ha grado d'ingresso 1.**

DEFINIZIONI

- **I nodi con grado di uscita 0 si dicono *foglie.***
- **La lunghezza del cammino dalla radice a un dato nodo si dice *livello* di quel nodo.**
- **La lunghezza del cammino più lungo dalla radice a una foglia si dice *altezza* dell'albero.**
- **Se un arco collega il nodo α al nodo β , il nodo α si dice *nodo padre* di β, il quale è detto *nodo figlio* (o *discen-dente diretto*) di α.**



Poiché *in un albero* il grado d'ingresso di ogni nodo è noto a priori, in luogo di ***"grado di uscita"*** si dice spesso semplicemente ***"grado"***.

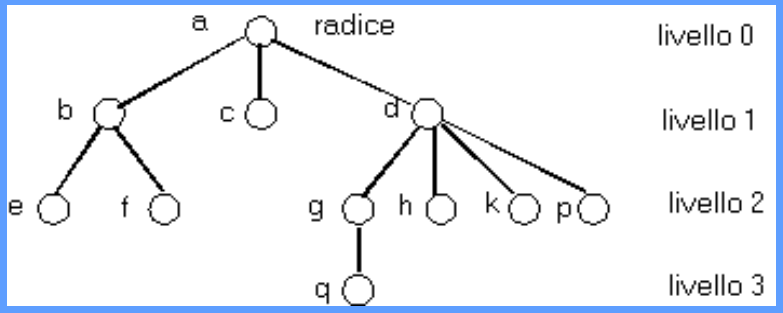
GRAFI e ALBERI

Un *albero* è un *grafo orientato aciclico* tale che

- **esiste un nodo (*radice*) con grado d'ingresso 0**
- **ogni altro nodo ha grado d'ingresso 1.**

CONSEGUENZE

- **esiste *esattamente un cammino* (semplice) dalla radice a qualsiasi altro nodo.**
- **tranne la radice, tutti i nodi hanno *esattamente un padre***
- **un padre può avere 0 o più figli**
- **tra i figli di un nodo esiste una *relazione d'ordine* che distingue il 1° nodo, il 2° nodo, etc** (disegnati solita-mente da sinistra a destra).



ALBERI

PROPRIETÀ SALIENTI

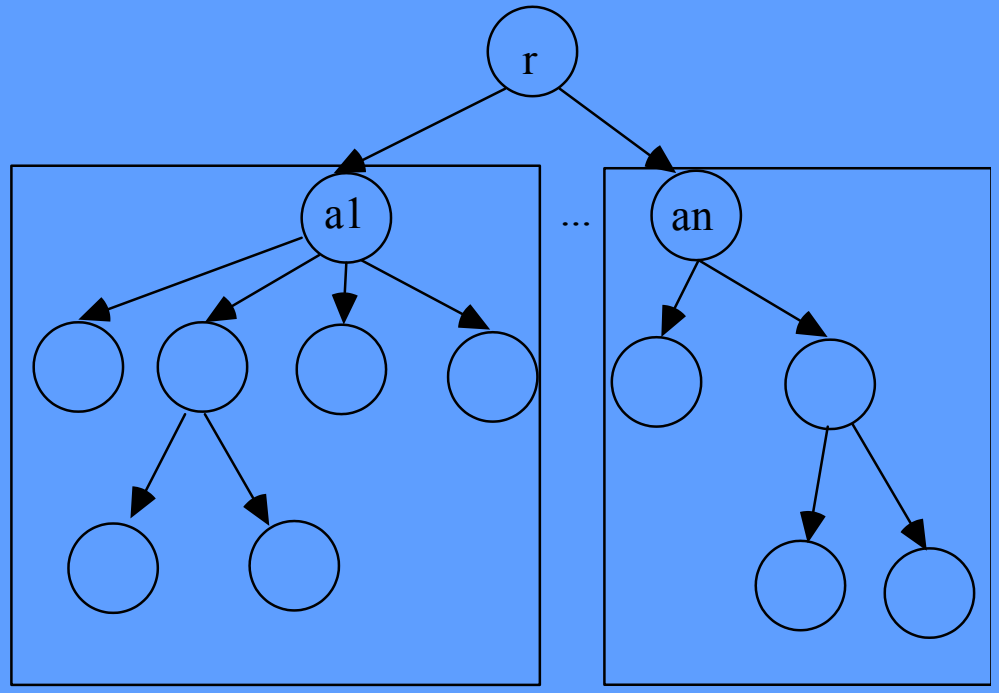
- Un albero rappresenta una *relazione d'ordine parziale* sull'insieme A
- Un albero può essere usato per rappresentare *relazioni gerarchiche* fra entità di A
- I *discendenti* di un dato nodo X sono calcolati come *chiusura transitiva* della relazione "figlio di".

CATEGORIE DI ALBERI

- Gli alberi si possono distinguere in base *al numero massimo di figli* che *ogni* loro nodo può avere.
- In un *albero binario*, ogni nodo ha *al più due figli* (ma può averne uno solo o nessuno)
- In un *albero ternario*, ogni nodo ha *al più tre figli* (ma può averne meno, o nessuno)
- …
- In un *albero N-ario*, ogni nodo ha *al più N figli*.

ALBERI come STRUTTURE RICORSIVE

- **Esclusa la radice, in un albero N-ario i nodi possono essere ripartiti in *N insiemi disgiunti***
- **Ciascuno di tali sottoinsiemi comprende *un figlio della radice più tutti (e soli) i suoi discendenti***
- **Ognuno di questi sottoinsiemi individua un *(sotto)albero***



VISITA DI UN ALBERO

- **Con il termine *visita* si intende *percorrere l'albero* secondo un qualche criterio, in modo da passare *una e una sola volta in ogni nodo.***
- **Data la natura *intrinsecamente non sequenziale* dell'albero, non esiste un concetto di "sequenza naturale" degli elementi** (come esisteva nelle liste).
- **Occorre definire *uno o più criteri di visita* che assicurino comunque di visitare, *ciascuno una sola volta,* tutti gli elementi dell'albero.**

VISITA DI UN ALBERO

Dato che un albero è definito come una struttura caratterizzata da

- **un elemento**
- **N (sotto)alberi**

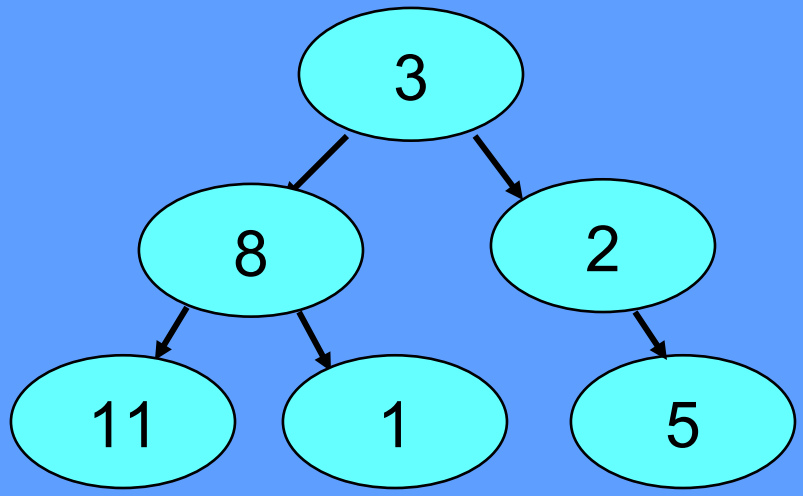
vi sono due criteri "naturali" di visita:

- ***visita in ordine anticipato (preorder):***
  prima la radice, poi tutti i sottoalberi
- ***visita in ordine posticipato (postorder):***
  prima tutti i sottoalberi, poi la radice.

ESEMPIO

Nel caso dell'albero a fianco:

- ***visita in preorder*** (prima la radice, poi tutti i sottoalberi):
  {3, *sinistro*, *destro* } =
  {3, {8,{11,1}}, {2,{5}}}
- ***visita in postorder***
  (prima tutti i sottoalberi, poi la radice):
  {*sinistro*, *destro*, 3} =
  {{{11,1},8}, {{5},2}, 3}



VISITA IN AMPIEZZA

Un terzo criterio sfrutta la struttura *a livelli* dell'albero:

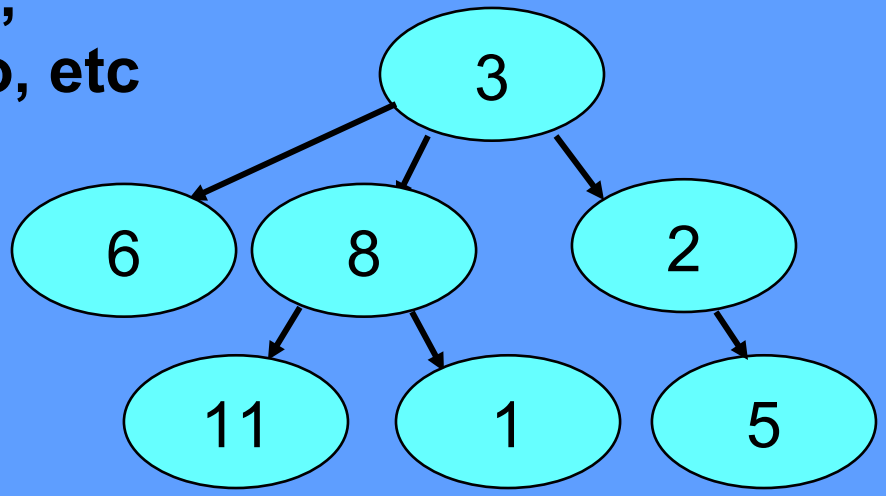
- **livello 0: la radice**
- **livello 1: i figli della radice**
- **livello 2: i figli dei figli della radice ...**

Visita in ampiezza (breadth-first):

- **prima la radice (livello 0)**
- **poi tutti i nodi del I° livello,**
- **poi tutti quelli del II° livello, etc**

Nel caso dell'albero a fianco:

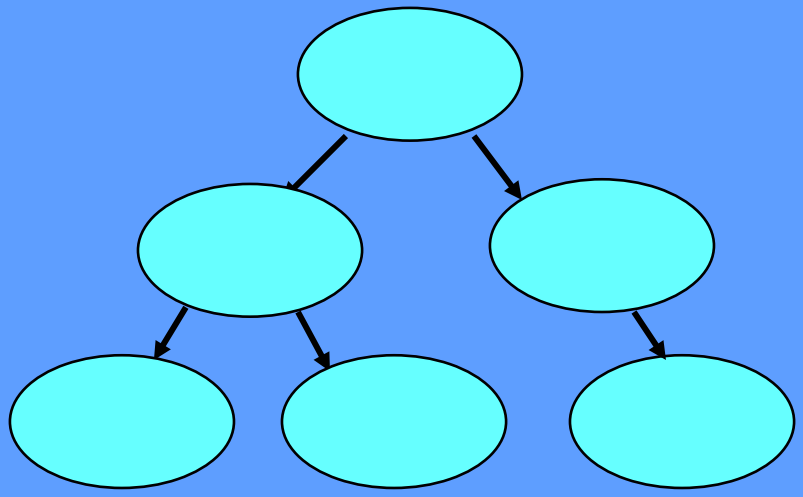
{3, *livello1*, *livello2*} =
{3, {6, 8, 2}, {11,1,5} }



ALBERI BINARI

Il caso più tipico di albero è l'albero *binario*

- **un elemento**
- **due (sotto)alberi figli, *sinistro* e *destro***



VISITA DI UN ALBERO BINARIO

Nel caso particolare di un albero binario, che ha *due* sottoalberi, è possibile definire un ulteriore criterio "naturale" di visita:

visita in ordine (inorder)

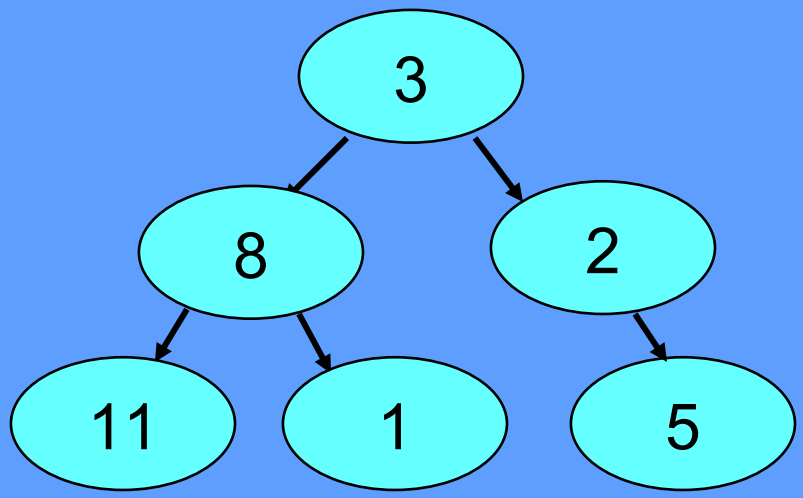
- **prima il sottoalbero di sinistra,**
- **poi la radice,**
- **poi il sottoalbero di destra.**

Questo criterio *ha senso solo per un albero binario,* in quanto per un albero generico con N figli esistono N possibili visite "in ordine", nessuna più significativa delle altre.

ESEMPIO

Nel caso dell'albero a fianco:

- ***visita in ordine*** (prima il sottoalbero sinistro, poi la radice, poi il destro): **{*sinistro*, 3, *destro* } = {{{11},8,{1}}, 3, {2,{5}}}**



ALBERI BINARI DI RICERCA

- **Una *albero binario di ricerca (binary search tree)* è un albero binario i cui elementi rispettano tutti una data *relazione d'ordine***
  - gli elementi devono dunque appartenere a un dominio in cui esista una relazione d'ordine totale
- **In particolare, detto R il valore della radice:**
  - **il sottoalbero di sinistra contiene solo elementi *minori o uguali* a R**
  - **il sottoalbero di destra contiene solo elementi *maggiori* di R.**

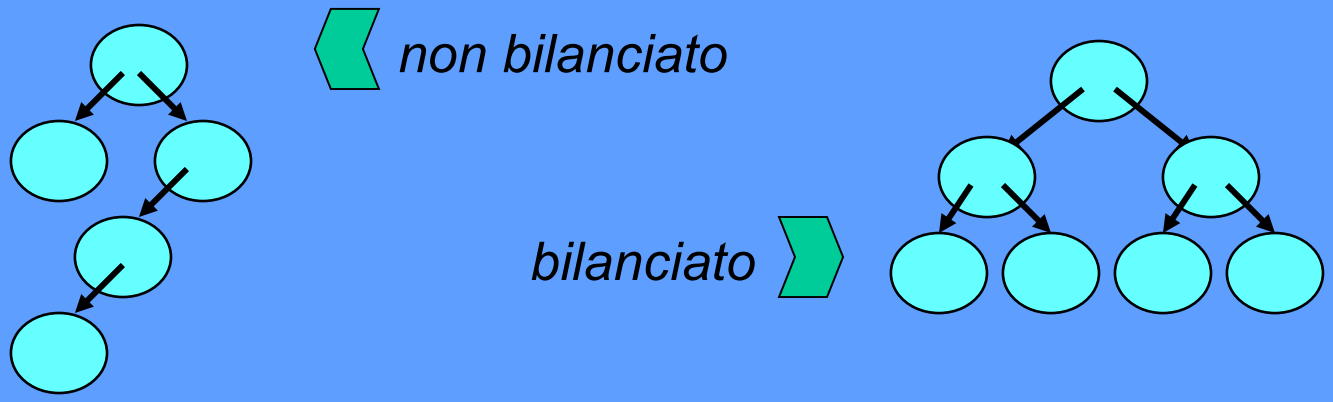
ALBERI BINARI DI RICERCA

Un albero binario di ricerca *velocizza notevolmente la ricerca di un elemento,* perché *esclude metà albero a ogni confronto*

- **l'esito del confronto dice da che parte sta l'elemento:**
  - **nel sottoalbero di sinistra, se l'elemento cercato è *minore* della radice**
  - **nel sottoalbero di destra, se l'elemento cercato è *maggiore* della radice.**
- **dopo K confronti, ha operato K dimezzamenti → ha esplorato uno spazio di 2^K elementi**
- ***Ricerca binaria*: per esplorare N elementi occorrono al più $K = \log_2 N$ confronti.**

ALBERI BILANCIATI

- **Un albero i cui elementi sono distribuiti uniformemente fra i sottoalberi si dice *bilanciato***
- **Molti algoritmi funzionano meglio con alberi bilanciati.**
- **Esistono algoritmi per bilanciare alberi non bilanciati.**



ALBERI: VALORI o CONTENITORI?

ALBERI COME VALORI

- **Un albero è una ennupla** costituita da un elemento e da N (sotto)alberi
- Il valore ***alberovuoto*** è dato a priori.

CONSEGUENZE

- Un albero non è un contenitore
- Non si può *cambiare* un albero esistente
- È un approccio sicuro (non si modifica mai niente)
- Ma non è sempre efficiente

ALBERI COME CONTENITORI

- **Un albero è un grafo aciclico di *contenitori* di elementi**

CONSEGUENZE

- Un albero è un contenitore
- *Si può* cambiare un albero già esistente
- È un approccio efficiente (non si ricostruisce ciò che già esiste)
- Ma può essere pericoloso in presenza di *structure sharing* e *aliasing*

ALBERI COME VALORI: PRIMITIVE

Costruttore

- **consTree**: elType × tree × tree → tree

(In Java può diventare il costruttore della classe)

Selettori

- **root**: tree → elType
- **left**: tree → tree
- **right**: tree → tree

Predicati

- **isEmptyTree**: tree → boolean

In Java sono metodi

ALBERI COME VALORI: PRIMITIVE

Costruttore

- **consTree**: elType × tree × tree → tree

(In Java diventare il costruttore della classe)

Dati un elemento e due alberi già esistenti, T1 e T2, il costruttore *crea un nuovo albero* avente tale elemento come radice, e i due alberi T1 e T2 come figli (nell'ordine, sinistro e destro).

Caso particolare:
uno o entrambi gli alberi T1 e T2 possono essere la costante *alberovuoto.*

- **isEmptyTree**: tree → boolean

ALBERI COME VALORI: PRIMITIVE

Costruttore

- **consTree**: elType × tree × tree → tree
  (costruisce un albero dati un elemento e due alberi figlio, sinistro e destro, rispettivamente)

Selettori

- **root**: tree → elType
- **left**: tree → tree
- **right**: tree → tree

Dato un albero, *seleziona l'elemento radice*

In Java sono metodi

Predicati

- **isEmpty**: tree → boolean

Dato un albero, *selezionano rispettivamente il figlio di sinistra o quello di destra*.

ALBERI COME VALORI: PRIMITIVE

Costruttore

- **consTree**: elType × tree × tree → tree

(In Java può diventare il costruttore della classe)

Selettori

- **root**: tree → elType
-
-

a sono
li

Dato un albero, risponde sì o no alla domanda:
"È vuoto"?

Predicati

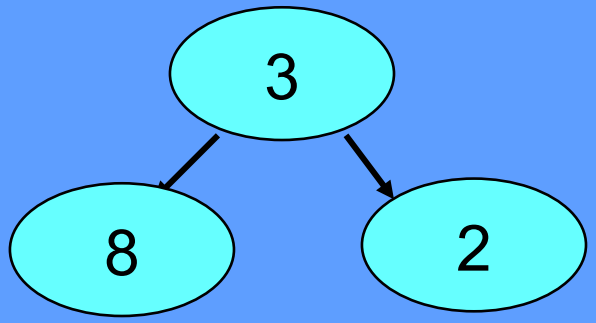
- **isEmptyTree**: tree → boolean

ALBERI COME VALORI: COSTRUZIONE

- **Un *albero come valore* è definito in modo *induttivo***
  - **esiste una costante *alberovuoto* data a priori**
  - **ogni altro albero si ottiene anteponendo un nuovo elemento radice a due alberi pre-esistenti.**
- Ad esempio, l'albero illustrato a fianco si ottiene come:

```
Tree t = new Tree(3,
  new Tree(8,ET,ET),
  new Tree(2,ET,ET)
);
```

(*ET* indica l'albero vuoto)



ALBERI COME VALORI: ALGORITMI

- **Se gli alberi sono definiti in modo *induttivo*, gli algoritmi che li manipolano sono naturalmente espressi in modo *ricorsivo.***
- **Tali algoritmi sono *indipendenti dalla rappresentazione*** (ovviamente, cambia la sintassi)
  - **non è rilevante come gli alberi siano realizzati**
  - **non importa il linguaggio (C, Pascal, Java…)**
- **Gli algoritmi si basano solo sulle primitive** (consTree, root, left, right, isEmptyTree).

ESEMPIO

Conteggio degli elementi di un albero

Algoritmo

- **se l'albero è vuoto, gli elementi sono zero**
- **altrimenti, gli elementi sono 1 (la radice) + quelli del figlio sinistro + quelli del figlio destro**

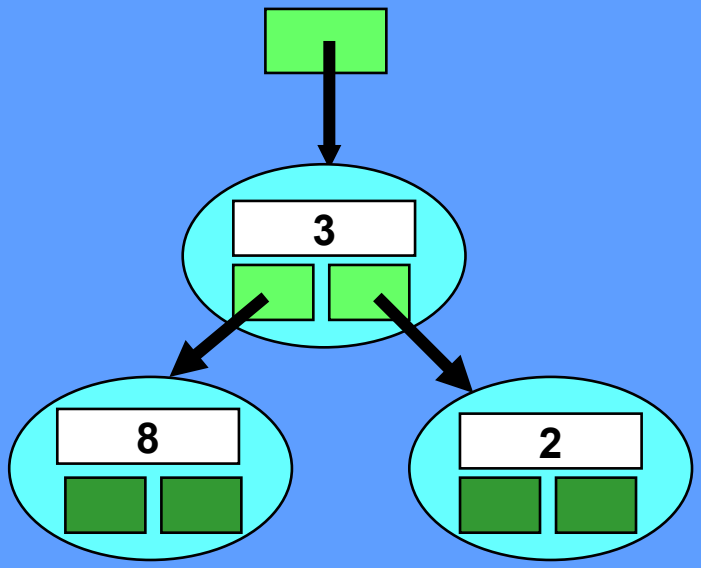
Specifica (pseudo-C):

```
count(T) =
 0,     se T == alberovuoto
 1 + count(left(T)) + count(right(T)),
altrimenti
```

In Java, **`count`** sarà un metodo.

ALBERI: RAPPRESENTAZIONE

- **Un *albero* è quasi sempre rappresentato tramite *nodi concatenati.***
- **Ogni *nodo* è composto di *tre parti*: un *valore* e due *"riferimenti" ai figli*** (nel caso binario, i sottoalberi sinistro e destro).
- **La *rappresentazione fisica* è solitamente basata su puntatori (o riferimenti) e allocazione dinamica.**



ALBERI: RAPPRESENTAZIONE

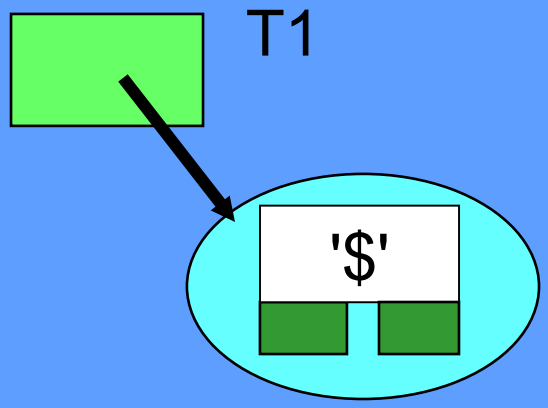
- **In questo contesto, l' *albero vuoto* è rappresentata dall' *assenza di nodi***

  - **un puntatore (o riferimento) nullo**
  - **un oggetto specifico (l' *alberovuoto*)**
  - **...**

ALBERI: RAPPRESENTAZIONE

- Se gli alberi sono valori, *una volta creata una struttura, non la si tocca mai più*
- **Si possono solo *creare nuove strutture***

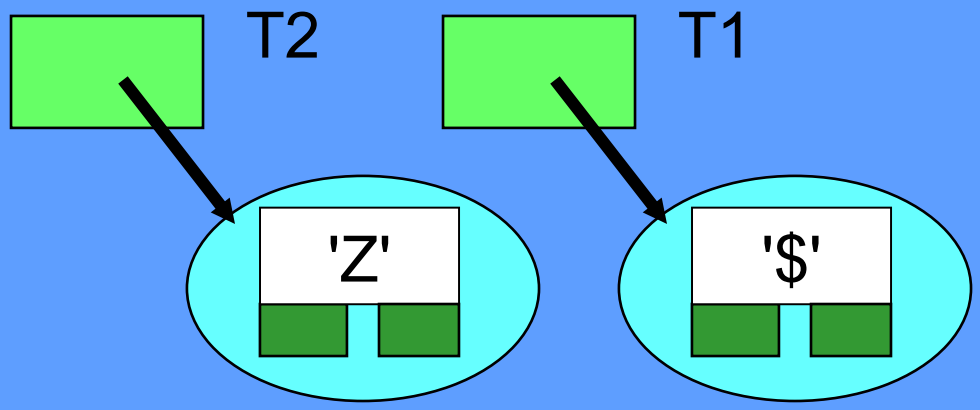
```
T1 = consTree('$', ET, ET);
```



ALBERI: RAPPRESENTAZIONE

- Se gli alberi sono valori, *una volta creata una struttura, non la si tocca mai più*
- **Si possono solo *creare nuove strutture***

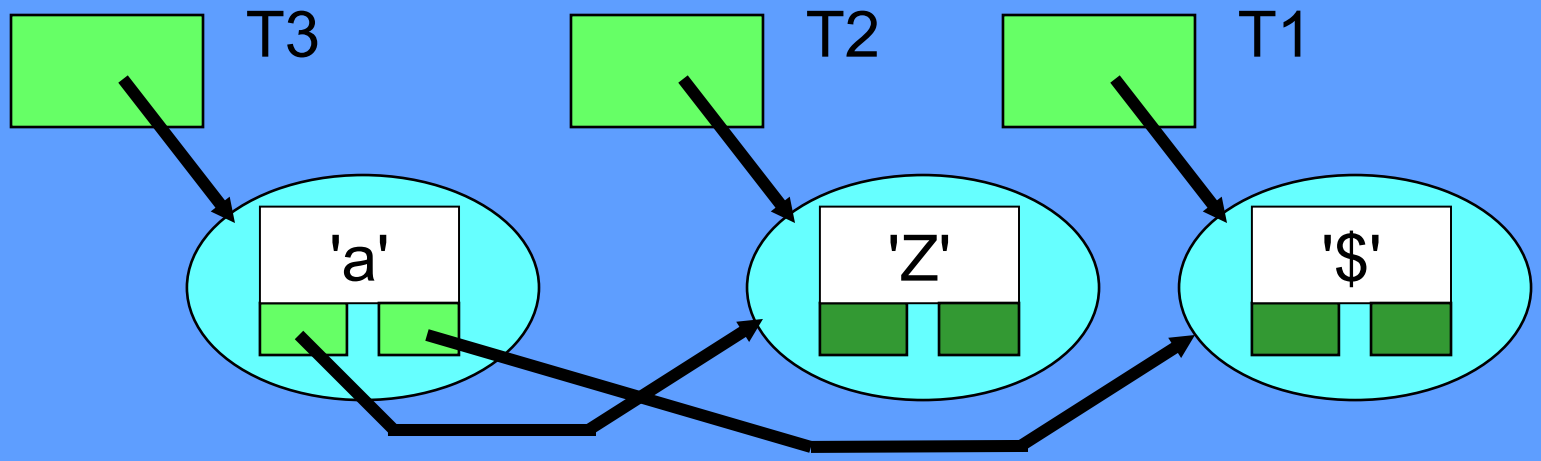
```
T1 = consTree('$', ET, ET);
T2 = consTree('Z', ET, ET);
```



ALBERI: RAPPRESENTAZIONE

- Se gli alberi sono valori, *una volta creata una struttura, non la si tocca mai più*
- **Si possono solo *creare nuove strutture***

```
T1 = consTree('$', ET, ET);
T2 = consTree('Z', ET, ET);
T3 = consTree('a', T2, T1);
```



STRUCTURE SHARING e ALIASING

Se gli alberi sono valori:

- **il costruttore opera normalmente con *structure sharing***
  - **si limita a *creare il nuovo nodo radice* e a *collegarlo* con gli alberi pre-esistenti**
- **un assegnamento `NewT = T` crea *aliasing***
  - **ci si limita a *creare un nuovo riferimento* all'albero già esistente, senza duplicare alcunché.**
- **sono pratiche *sicure* perché le strutture dati *non vengono mai modificate*.**

RIFLESSIONI

- **L'approccio funzionale (alberi come valori) assicura assenza di *side effects* anche nel caso di structure sharing e aliasing...**
- **… ma può essere più costoso**
  - **per cambiare il valore di un nodo occorre *ricostruire una parte dell'albero***
  - **si produce molto facilmente *garbage* (non si distrugge mai nulla) → *inefficiente* se il linguaggio non dispone di un *garbage collector*** *(come nel caso del C)*

ALGORITMI DI VISITA

- **Data la natura *intrinsecamente non sequenziale* dell'albero, non è possibile scorrere uno a uno gli elementi con un ciclo**
  - **non c'è una nozione "naturale" o "implicita" di "elemento successivo"**
- **Occorre adottare un *approccio ricorsivo***
  - **si opera sulla radice (prima, in mezzo, o dopo)**
  - **si richiama ricorsivamente la visita sui sotto-alberi figli.**

VISITA IN ORDINE ANTICIPATO

- **in C:**

```
void preOrder(tree T) {
 if (T == alberovuoto) return;
 printElement(root(T));
 preOrder(left(T)); preOrder(right(T));
}
```

- **in Java** ***(approccio a classi, o implem. di interfaccia):***

```
public class Tree {  // albero non vuoto
 …
 public void preOrder() {
 System.out.println(root());
 left().preOrder(); right().preOrder();
}
```

VISITA IN ORDINE POSTICIPATO

- **in C:**

```
void postOrder(tree T) {
 if (T == alberovuoto) return;
 postOrder(left(T)); postOrder(right(T));
 printElement(root(T));
}
```

- **in Java** ***(approccio a classi, o implem. di interfaccia):***

```
public class Tree {  // albero non vuoto
 …
 public void postOrder() {
  left().postOrder(); right().postOrder();
  System.out.println(root());
 }
```

VISITA IN ORDINE (solo alberi binari)

- **in C:**

```
void inOrder(tree T) {
 if (T == alberovuoto) return;
 inOrder(left(T)); printElement(root(T));
 inOrder(right(T));
}
```

- **in Java** ***(approccio a classi, o implem. di interfaccia):***

```
public class Tree {  // albero non vuoto
 …
 public void inOrder() {
 left().inOrder(); System.out.print(root());
 right().inOrder();
}
```

VISITE & ALGORITMI

- **Poiché la visita ricorsiva *è il solo modo per scorrere uno ad uno tutti gli elementi di un albero,* tutti gli algoritmi che operano su alberi sono una "variazione sul tema" di uno degli algoritmi di visita.**
- **<u>Cambia solo l'operazione da fare sulla radice</u>:**
  - **non più una semplice stampa (print)...**
  - **ma confronti, test ...**

ESEMPIO

Ricerca di un elemento in un albero

- **se l'albero è vuoto, l'elemento non c'è**

  altrimenti (si sfrutta la visita in pre-order)
- **se tale elemento è la radice, lo si è trovato, altrimenti va cercato nei sottoalberi figli.**

Esempio Java *(caso albero non vuoto)*

```
public boolean member(Object e) {
 if (e.equals(root())) return true;
 if (left().member(e)) return true;
 else return right().member(e);
}
```

ESERCIZIO

Scrivere un algoritmo che, dato un albero di interi, conti gli elementi uguali a un dato X.

- Prendiamo un algoritmo di visita e, al posto della stampa, inseriamo il test richiesto
- Restituiamo il risultato del conteggio nei figli, *aumentato di 1 se il test è risultato positivo.*

Esempio Java *(caso albero non vuoto)*

```
public int countEquals(Object x) {
  return left().countEquals(x) +
         right().countEquals(x) +
         (root().equals(x)>0? 1 : 0);
}
```

ALBERI BINARI DI RICERCA

Come creare un albero binario di ricerca?

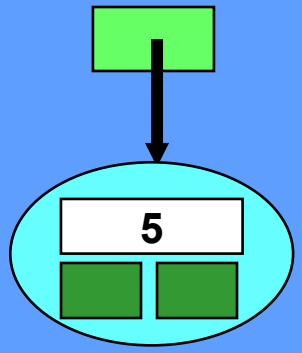
- **Aggiungendo via via i nuovi elementi *in modo da rispettare la relazione d'ordine***

Nel caso di alberi come valori, si crea *un nuovo albero*

- **se l'elemento è minore o uguale alla radice, l'elemento va inserito nel sottoalbero di sinistra**
- **se è maggiore, va nel sottoalbero di destra**
- **quando si arriva a un sottoalbero vuoto, si aggiunge l'elemento come nuova foglia.**

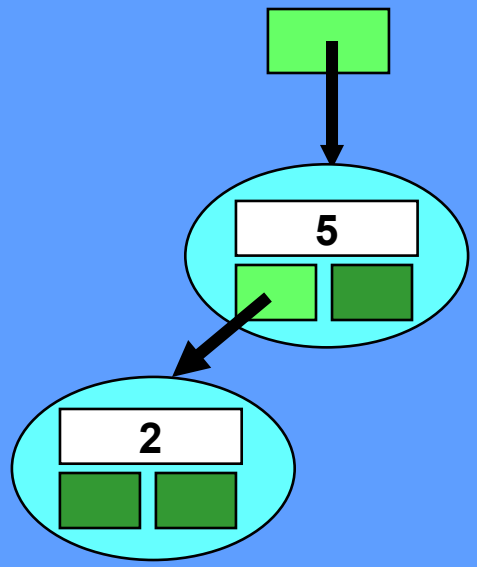
ESEMPIO (1/4)

```
BinSearchTree t =
   new BinSearchTree(5);
```



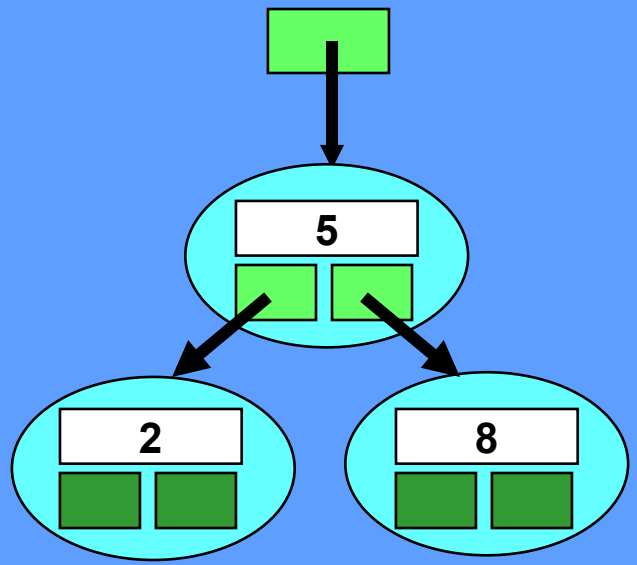
ESEMPIO (2/4)

```
BinSearchTree t =
   new BinSearchTree(5);
t = new BinSearchTree(2,t);
```



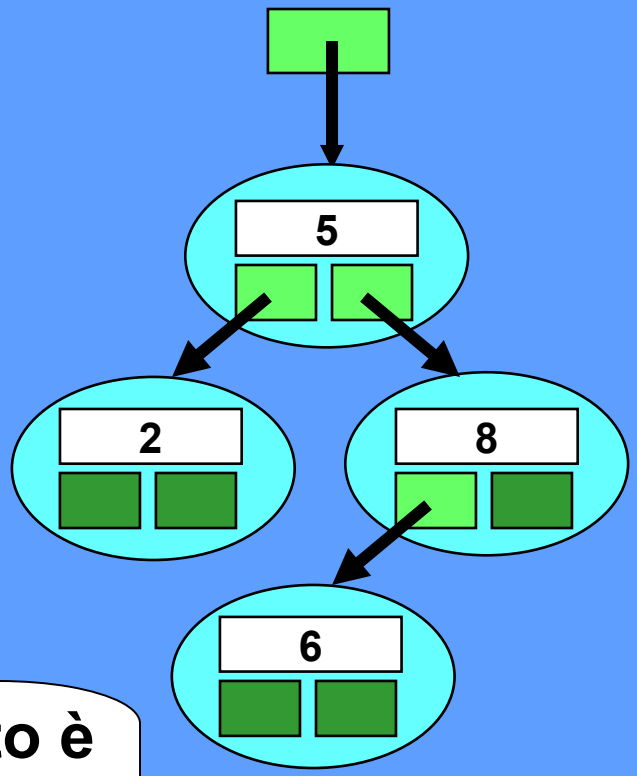
ESEMPIO (3/4)

```
BinSearchTree t =
   new BinSearchTree(5);
t = new BinSearchTree(2,t);
t = new BinSearchTree(8,t);
```



ESEMPIO (4/4)

```
BinSearchTree t =
   new BinSearchTree(5);
t = new BinSearchTree(2,t);
t = new BinSearchTree(8,t);
t = new BinSearchTree(6,t);
...
```



Come diventa l'albero se si inseriscono nell'ordine:

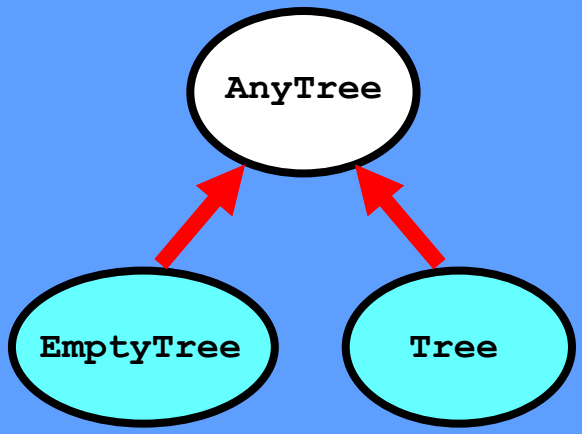
- **4, 3, 7, 9**
- **3, 4, 9, 7**
- **... ?**

L'ordine di inserimento è rilevante: ordine diverso implica albero diverso

ALBERI COME VALORI IN JAVA

Si pongono problemi (e opzioni) analoghi al caso delle liste

- **progetto basato su classi**



- **progetto basato su interfacce**

. . . . ?

La classe `AnyTree` è astratta

- dichiara tutte le operazioni indipendenti dalla rappresentazione, più il metodo `isEmpty`
- non dichiara le primitive `root`, `left` e `right` perché si applicano solo ad alberi non-vuoti.

La classe derivata `EmptyTree` è concreta

- implementa tutte le operazioni *nel caso specifico dell'albero vuoto*
- definisce `isEmpty` in modo che restituisca `true`

JAVA: GLI ALBERI PREDEFINITI

Il package `java.util` definisce:

- ***interfacce*** **che rappresentano** ***tipi di dati astratti di contenitori*** (cfr. **`SortedSet`**)
- ***classi*** **che forniscono** ***specifici contenitori*** **concreti** (cfr. **`TreeSet`**, che usa **`TreeMap`**)

I metodi principali (`add`, `remove`, `contains`, `isEmpty`, …) sono introdotti da `Collection` e specializzati dalle varie classi.